MONGO DB

Lezione IV

RIPASSO

RESOCONTO ESERCIZIO

AND THE WINNING TEAM IS

Voto basato sull'esercizio!

1 - Scobydoobydoo - 3 pt

2 - Jenkis Khan - 2 pt

3 - I Pavesini - 1 pt

Non qualificati - I Budini, i Mongolini

Classifica

1 - Scobyboodidoo - 7 pt

2 - I Mongolini / Jenkis Khan - 5 pt

3 - I Pavesini / I Budini - 3 pt

Atomicità

Cosa significa Atomicità quando si parla di operazioni in un Database?

Poniamo un esempio:

Quando inseriamo un documento (InsertOne()) questo elemento verrà salvato nella sua interezza OPPURE (in caso di errore) questo non verrà salvato neppure in parte.

Questo ci indica che l'operazione di insertOne ha una sua atomicità.

"Un'operazione atomica, in informatica, consiste in un'operazione di esecuzione indivisibile dal punto di vista logico."

O l'operazione ha successo o non ha successo in questo caso. Bisogna però stare attenti a come Mongo interpreta questa operazione.

Atomicità

Riprendiamo un esempio simile utilizzando insertMany().

Poniamo l'esempio di 5 documenti da inserire. Se arrivati al numero 3 questo va in errore cosa accade?

Quanti documenti abbiamo inserito? Dove si trova l'atomicità dell'operazione?

In questo esempio i documenti inseriti sono soltanto 2 (in caso di ordered: true).

Il numero 3 viene escluso in quanto ha generato errore.

Questo ci fa notare che l'atomicità del comando è contestualizzata al documento e non all'intera operazione.

Abbiamo un modo per garantire atomicità sull'intera operazione?

Transaction

Per garantire l'atomicità su più operazioni si ricorre alle transaction.

Questa operazione ci garantisce che tutti i comandi che lanciamo vengano eseguiti nel loro insieme o che falliscano nel loro insieme.

Una transazione atomica è costituita dal seguente schema →

1. Start transaction
2. Operazioni da eseguire
3. Direttive commit o roll back ossia di convalida o di disfacimento dell'intera transazione.
4. End transaction

Transaction

Per garantire l'atomicità su più operazioni si ricorre alle transaction.

Questa operazione ci garantisce che tutti i comandi che lanciamo vengano eseguiti nel loro insieme o che falliscano nel loro insieme.

Una transazione atomica è costituita dal seguente schema →

1. Start transaction
2. Operazioni da eseguire
3. Direttive commit o roll back ossia di convalida o di disfacimento dell'intera transazione.
4. End transaction

Transaction

Come avviene questa operazione in pratica?

Si apre una connessione, un “proxy” del nostro database. Esempio per node.js:

```
const client = new MongoClient(uri);
await client.connect();
const session = client.startSession();
```

```
try {
   await session.withTransaction(async () => {
     const coll1 = client.db('mydb1').collection('foo');
     const coll2 = client.db('mydb2').collection('bar');

     // Important:: You must pass the session to the operations
     await coll1.insertOne({ abc: 1 }, { session });
     await coll2.insertOne({ xyz: 999 }, { session });
   });
 } finally {
   await session.endSession();
   await client.close();
 }
```

Il codice all’interno può effettuare tutte le operazioni sui dati ma il database verrà effettivamente aggiornato al termine dell’intera transaction.

Aggregate

Quando si ha a che fare con una gran mole di dati – e MongoDB è nato proprio per questo scopo – una delle funzionalità più importanti è l'aggregazione, che permette di avere una visione globale dei dati e di ricavarne statistiche.

L'aggregazione è oggi una funzionalità essenziale in ambiti come Business Intelligence e Big Data.

MongoDB supporta due meccanismi per effettuare aggregazioni. Uno è quello storicamente più noto in NoSQL: MapReduce. Di recente, però, è stato introdotto anche l'Aggregation Framework.

Quest'ultimo è il meccanismo raccomandato, sia per la sua semplicità ma anche per ragioni di performance.

Andremo a utilizzare quest'ultimo.

Aggregation Framework

Il comando preposto a questa funzione è aggregate().

Questo comando accetta come parametro una pipeline di operatori, cioè un elenco ordinato di operatori di aggregazione.

L'effetto è che la collezione dei documenti passa tra i vari stadi della pipeline.

Per avere le migliori performance è quindi importante mettere all'inizio della pipeline le operazioni di filtraggio in modo da far passare meno elementi possibili attraverso il resto della pipeline.

Match Stage

Vediamo il primo operatore, $match

Questo ci permette di effettuare un filtro sulla base dati, esattamente come nel find()

```
db.person.aggregate( [
    { $match: { gender: "female" } }
])
```

Match prende in input un documento

Group Stage

L'operatore $group separa i documenti in gruppi usando una "group key" come discriminante. L'output è un singolo documento per ciascuna chiave unica.

Possiamo poi passare degli accumulatori che ci permettono di effettuare operazioni su questi dati.

```
db.person.aggregate( [
    { $group: { _id: { gender: "$gender"}, ← chiave
                        totalPerson: {$sum: 1} ← accumulatori
                        } }
])
```

Questi accumulatori vi ricordano qualcosa dalle lezioni dello scorso anno?

Project Stage

L'operatore $project passa i documenti allo stage successivo della pipeline con i campi richiesti. Questi possono essere campi presi dal documento, campi nuovi o computati.

```
db.person.aggregate( [
    { $project: { _id: 0,
                  gender: 1,
                  code:"TD",
                  fullname: {$concat: ["$name.first", " ", "$name.last"]}
                } }
])
```

MONGO DB

Lezione IV

Esercizio I Singolo

Esercizio I - Singolo

Usando la collection Person esegui una aggregation che restituisca la media dell'età con divisione per genere e nazionalità.

Questo solo per le persone maggiorenni.

Devi usare sia match che group che project.

Group VS Project

$group

Relazione 1:molti

Somme, conteggi, media, costruzione di array

$project

Relazione 1:1

Includi/escludi campi, trasformazione dei campi dentro il singolo documento.

Push

L'operatore $push permette di aggiungere un valore ad un array.

Questo, nell'aggregate, può essere usato per unire il contenuto array nestati.

```
db.friends.aggregate( [
    { $group: { _id: "$age", ← chiave - NB : ci sono 2 modi per indicarla
                    allHobbies: {$push: "$hobbies"}
                    } }
])
```

Questo però va a creare un array di array. Come possiamo fare per aggiungere semplicemente i valori?

Unwind Stage

L'operatore $unwind permette di "esplodere" il contenuto di un array.

```
db.friends.aggregate( [
    { $unwind: "$hobbies"}
])
```

Se osserviamo l'output vediamo che ogni documento è stato replicato per ogni elemento dell'array, al cui posto ora si trova un singolo elemento.

Possiamo ora usare $group e $push per creare un array di valori. Però non abbiamo ancora eliminato i duplicati!

addToSet

L'operatore $addToSet permette di aggiungere un valore ad un array a meno che questo non sia già presente.
Vediamolo insieme a unwind:

```
db.friends.aggregate( [
     { $unwind: "$hobbies"},
    { $group: { _id: "$age",
                        allHobbies: {$addToSet: "$hobbies"}
                        } }
])
```

push e addToSet sono entrambi utili per la manipolazione di array, semplicemente utilizzano una logica differente!

MONGO DB

Lezione IV

Esercizio II Singolo

Esercizio II - Singolo

Usando quanto abbiamo visto insieme oggi e nelle lezioni precedenti , tramite la collection Friends crea una pipeline che restituisca id, nome e punteggio massimo di ogni esame per ogni persona.

Il risultato sarà quindi simile a questo:

```
{
        "_id": objectId,
    "name": nome,
    "maxScore": valore massimo esame.
}
{
        "_id": objectId,
    "name": nome,
    "maxScore": valore massimo esame.
}
{
        "_id": objectId,
    "name": nome,
    "maxScore": valore massimo esame.
}
```

Bucket Stage

L'operatore $bucket permette di dividere il contenuto in gruppi, impostando "confini" e l'output desiderato.

```
db.person.aggregate( [
    { $bucket: {
                    groupBy: "$dob.age",
                    boundaries:[0,18,30,50,80],
                     output:{
                       averageAge: {$avg: "$dob.age"},
                       names:{$push:"$name.first"}
                    }}])
```

Questo operatore ci permette una suddivisione dei dati molto dettagliata per le più svariate esigenze

Auto Bucket

Possiamo utilizzare l'operatore $bucketAuto per lasciare a Mongo la creazione dei limiti. Mongo cercherà di creare la distribuzione più uniforme.

```
db.person.aggregate( [
    { $bucketAuto: {
                    groupBy: "$dob.age",
                    buckets:5, ← numero di bucket da creare
                     output:{
                       averageAge: {$avg: "$dob.age"},
                       names:{$push:"$name.first"}
                    }}])
```

L'algoritmo non è perfetto, è preferibile sempre impostare i valori manualmente. Può essere però utile in una prima fare per determinare i "macrovalori"

Skip, Limit e Out Stage

Esistono altri stage che tratteremo in maniera più rapida:

$skip: Permette di "saltare" un determinato numero di documenti nella restituzione

$limit: Permette di limitare il numero di documenti restituito.

Attenzione! Questi stage se usati in combinazione con $sort possono dare risultati differenti in base all'ordine di utilizzo! Mentre nel find il sort è indifferente dalla posizione, qui stiamo trattando di pipeline che vengono eseguite una di seguito all'altra!

$out: Crea una collection con il risultato della pipeline precedente. Se la collection già esiste la sovrascrive

Pipeline e Index

Ricorda bene: gli indici esistono solo nella prima pipeline all'interno di un'aggregate!

Tienilo a mente per creare istruzioni rapide e performanti!

MONGO DB

Lezione IV

Esercizio III Singolo

Esercizio III - Singolo

Usando quanto abbiamo visto insieme oggi e nelle lezioni precedenti , tramite la collection Person crea una pipeline che restituisca le venti persone più vecchie, per fasce d'età decennali, partendo da quelle più vecchie di 30 anni, divise per genere.

Inserisci poi questi valori in una nuova collection

MONGO DB

Lezione IV

Esercizio IV di gruppo

Esercizio IV - Di gruppo

Create un DB contenente 3 collection:

Accounts, customers e transactions.

Usando i file allegati popolate le collection.

Create una pipeline per ottenere il numero totale delle transaction di ogni customer, con indicazione del nome. I customer devono essere divisi per età secondo gli intervalli che ritieni opportuni e che abbiano senso.

Create una pipeline che dia il totale del buy/sell di ogni transaction, raggruppata per account. Da questo estrai dagli account i prodotti con il maggior numero di acquisti e vendite.